J Cryptol (2019) 32:941–972
https://doi.org/10.1007/s00145-018-9301-4

Journal of
CRYPTOLOGY

Oblivious Network RAM and Leveraging Parallelism to Achieve Obliviousness

Dana Dachman-Soled
Department of Electrical and Computer Engineering, University of Maryland, College Park, USA
University of Maryland Institute for Advanced Computer Studies (UMIACS), College Park, USA
danadach@ece.umd.edu

Chang Liu
University of California, Berkeley, USA
liuchang@cs.umd.edu

Charalampos Papamanthou
Department of Electrical and Computer Engineering, University of Maryland, College Park, USA
University of Maryland Institute for Advanced Computer Studies (UMIACS), College Park, USA
cpap@umd.edu

Elaine Shi
Cornell University, Ithaca, USA
runting@gmail.com

Uzi Vishkin
Department of Electrical and Computer Engineering, University of Maryland, College Park, USA
University of Maryland Institute for Advanced Computer Studies (UMIACS), College Park, USA
vishkin@umiacs.umd.edu

Communicated by Alon Rosen.

Received 13 December 2016 / Revised 7 May 2018
Online publication 9 August 2018

Abstract. Oblivious RAM (ORAM) is a cryptographic primitive that allows a trusted CPU to securely access untrusted memory, such that the access patterns reveal nothing about sensitive data. ORAM is known to have broad applications in secure processor design and secure multiparty computation for big data. Unfortunately, due to a logarithmic lower bound by Goldreich and Ostrovsky (J ACM 43(3):431–473, 1996), ORAM is bound to incur a moderate cost in practice. In particular, with the latest developments in ORAM constructions, we are quickly approaching this limit, and the room for performance improvement is small. In this paper, we consider new models of computation in which the cost of obliviousness can be fundamentally reduced in comparison with the standard ORAM model. We propose the oblivious network RAM model of computation, where a CPU communicates with multiple memory banks, such that the adversary observes only which bank the CPU is communicating with, but not the address offset within each memory bank. In other words, obliviousness within each bank comes for free—either because the architecture prevents a malicious party from observing the address accessed within a bank, or because another solution is used to obfuscate

memory accesses within each bank—and hence we only need to obfuscate communication patterns between the CPU and the memory banks. We present new constructions for obliviously simulating general or parallel programs in the network RAM model. We describe applications of our new model in distributed storage applications with a network adversary.

Keywords. Oblivious RAM, Parallel-computing, PRAM model.

1. Introduction

Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky [19,20], allows a *trusted* CPU (or a trusted computational node) to obliviously access *untrusted* memory (or storage) during computation, such that an adversary cannot gain any sensitive information by observing the data access patterns. Although the community initially viewed ORAM mainly from a theoretical perspective, there has recently been an upsurge in research on both new efficient algorithms (c.f. [8,13,25,41,44,47,50]) and practical systems [9,11,12,24,34,39,42,43,48,52] for ORAM. Still the most efficient ORAM implementations [10,42,44] require a relatively large bandwidth blowup, and part of this is inevitable in the standard ORAM model. Fundamentally, a well-known lower bound by Goldreich and Ostrovsky states that in a balls-and-bins model, which encompasses all known ORAM constructions, any ORAM scheme with constant CPU cache must incur at least $\Omega(\log N)$ blowup, where N is the number of memory words, in terms of bandwidth and runtime. To make ORAM techniques practical in real-life applications, we wish to further reduce its performance overhead. However, since the latest ORAM schemes [44,47] have practical performance approaching the limit of the Goldreich–Ostrovsky lower bound, the room for improvement is small in the standard ORAM model. In this paper, we investigate the following question:

> *In what alternative, practically-motivated, models of computation can we significantly lower the cost of oblivious data accesses?*

We propose the network RAM (NRAM) model of computation and correspondingly, oblivious network RAM (O-NRAM). In this new model, one or more CPUs interact with M memory banks during execution. Therefore, each memory reference includes a *bank identifier*, and an *offset* within the specified memory bank. We assume that an *adversary cannot observe the address offset within a memory bank, but can observe which memory bank the CPU is communicating with*. In other words, obliviousness within each bank "comes for free". Under such a threat model, an oblivious NRAM (O-NRAM) can be informally defined as an NRAM whose observable memory traces (consisting of the bank identifiers for each memory request) do not leak information about a program's private inputs (beyond the length of the execution). In other words, in an O-NRAM, the sequence of bank identifiers accessed during a program's execution must be provably obfuscated.

1.1. *Distributed Storage with a Network Adversary*

Our NRAM models are motivated by the following application domain (and hence the name, network ORAM): Consider a scenario where a client (or a compute node) stores private, encrypted data on multiple distributed storage servers. We consider a setting where all endpoints (including the client and the storage servers) are *trusted*, but the network is an *untrusted* intermediary. In practice, trust in a storage server can be bootstrapped through means of trusted hardware such as the trusted platform module (TPM) or as IBM 4758; and network communication between endpoints can be encrypted using standard SSL. Trusted storage servers have also been built in the systems community [3]. On the other hand, the untrusted network intermediary can take different forms in practice, e.g., an untrusted network router or WiFi access point, untrusted peers in a peer-to-peer network (e.g., Bitcoin, TOR), or packet sniffers in the same LAN. Achieving oblivious data access against such a network adversary is precisely captured by our O-NRAM model.

1.2. *Background: The PRAM Model*

Two of our main results deal with the parallel RAM (PRAM) computational model, which is a synchronous generalization of the RAM computational model to the parallel processing setting. The PRAM computational model allows for an unbounded number of parallel processors with a shared memory spawned statically. Each processor may access any shared memory cell and read/write conflicts are handled in various ways depending on the type of PRAM considered:

- *Exclusive Read Exclusive Write (EREW) PRAM* A memory cell can be accessed by at most one processor in each time step.
- *Concurrent Read Exclusive Write (CREW) PRAM* A memory cell can be read by multiple processors in a single time step, but can be written to by at most one processor in each time step.
- *Concurrent Read Concurrent Write (CRCW) PRAM* A memory cell can be read and written to by multiple processors in a single time step. Reads are assumed to complete prior to the writes of the same time step. Concurrent writes are resolved in one of the following ways: (1) *Common*—all concurrent writes must write the same value; (2) *Arbitrary*—an arbitrary write request is successful; (3) *Priority*—processor id determines which processor is successful.

To realize a PRAM algorithm in practice, the algorithm must first be translated into standard code and then implemented on a particular architecture that supports the PRAM model. This is analogous to the case of algorithms in the RAM computational model, where various steps need to be taken under-the-hood in order to obtain an implementation for a specific architecture. The PRAM-On-chip project at UMD has demonstrated construction feasibility of the (so-called) XMT architecture, which supports the PRAM computational model [46]. Moreover, the work of Ghanim et al. [17] establishes that casting parallel algorithms using PARDO (the lockstep pseudo-code command used in PRAM textbooks to express parallelism) combined with a standard serial language (e.g., C) is all that is needed to get the same performance on XMT as the best manually optimized threaded code.

1.3. *Results and Contributions*

We introduce the oblivious network RAM model and conduct the first *systematic* study to understand the "cost of obliviousness" in this model. We consider running both *sequential* programs and *parallel* programs in this setting. We propose novel algorithms that exploit the "free obliviousness" within each bank, such that the obliviousness cost is significantly lower in comparison with the standard oblivious (parallel) RAMs. While we view our results as mainly theoretical, in many cases the concrete constants of our constructions are quite low. We therefore leave open the question of practically implementing our results and believe this is an interesting direction for future research. We give a summary of our results below.

First, in addition to the standard assumption that $N := N(\lambda)$ for polynomial $N(\cdot)$, where N is the total number of memory words and λ is security parameter, all our results require that $N \geq \lambda$. This holds in practical settings since λ is typically very small in comparison with the size of memory. Alternatively, we can view our results as being applicable to memory of *any* polynomial size N', but requiring a preliminary step of padding the memory up to size $N = N' + \lambda$.

Given the above assumption, we now discuss our results for the oblivious network RAM model. Observe that if there are only $O(1)$ number of memory banks, there is a trivial solution with $O(1)$ cost: just make one memory access (real or dummy) to each bank for each step of execution. On the other hand, if there are $\Omega(N)$ memory banks each of constant size, then the problem approaches standard ORAM [19,20] or OPRAM [7]. The intermediate parameters are therefore the most interesting. For simplicity, in this section, we mainly state our results for the most interesting case when the number of banks $M := M(\lambda) \in O(\sqrt{N})$, and each bank can store up to $O(\sqrt{N})$ words. In Sects. 3, 4 and 5, our results will be stated for more general parameter choices. We now state our results (see also Table 1 for an overview).

"Sequential-to-Sequential" Compiler First, we show that any RAM program can be obliviously simulated on a network RAM, consuming only $O(1)$ words of local CPU

Table 1. A systematic study of "cost of obliviousness" in the network ORAM model.

Setting	RAM to O-NRAM blowup	c.f. Best-known ORAM blowup
Sequential-to-sequential compiler		
$W =$ small	$\widehat{O}(\log N)$	$O(\log^2 N / \log\log N)$ [29]
$W = \Omega(\log^2 N)$	Bandwidth: $\widehat{O}(1)$	Bandwidth: $\widehat{O}(\log N)$ [47]
	Runtime: $\widehat{O}(\log N)$	Runtime: $O(\log^2 N / \log\log N)$ [29]
$W = \Omega(N^\epsilon)$	$\widehat{O}(1)$	$\widehat{O}(\log N)$ [47]
Parallel-to-sequential compiler		
$\omega(M \log N)$-parallel	$O(1)$	Same as standard ORAM
Parallel-to-parallel compiler		
$M^{1+\delta}$-parallel for any const $\delta > 0$	$O(\log^* N)$	Best known: poly log N [7] Lower bound: $\Omega(\log N)$

$W := W(\lambda)$ denotes the memory word size in # bits, $N := N(\lambda)$ denotes the total number of memory words, and $M := M(\lambda)$ denotes the number of memory banks. For simplicity, this table assumes that $M \in O(\sqrt{N})$, and each bank has $O(\sqrt{N})$ words. Like implicit in existing ORAM works [20,29], small word size assumes at least log N bits per word—enough to store a virtual address of the word

cache, with $\widehat{O}(\log N)$ blowup in both runtime and bandwidth, where—throughout the paper—when we say the complexity of our scheme is $\widehat{O}(f(N))$, we mean that for any choice of $h(N) = \omega(f(N))$, our scheme attains complexity $g(N) = O(h(N))$. Further, when the RAM program has $\Omega(\log^2 N)$ memory word size, it can be obliviously simulated on network RAM with only $\widehat{O}(1)$ bandwidth blowup (assuming non-uniform memory word sizes as used by Stefanov et al. in [43]). In comparison, the best-known (constant CPU cache) ORAM scheme has roughly $\widehat{O}(\log N)$ bandwidth blowup for $\Omega(\log^2 N)$ memory word size [47]. For smaller memory words, the best-known ORAM scheme has $O(\log^2 / \log\log N)$ blowup in both runtime and bandwidth [29].

"Parallel-to-Sequential" Compiler We demonstrate that parallelism can facilitate obliviousness, by showing that programs with a "sufficient degree of parallelism"—specifically, programs which have $P := P(\lambda) \in \omega(M \log N)$ number of operations (where λ is security parameter) that can be executed in parallel at each time step—can be obliviously simulated in the network RAM model with only $O(1)$ blowup in runtime and bandwidth. Here, we consider parallelism as a property of the program, but are not in fact executing the program on a parallel machine. The overhead stated above is for the sequential setting, i.e., considering that both NRAM and O-NRAM have a single processor. Our compiler works when the underlying PRAM program is in the EREW, CREW or common/arbitrary/priority CRCW model. Note that a PRAM-supporting architecture is not required for realization of this result, since the final compiled program is executed in a *sequential* setting. We use only the fact that the underlying algorithm can be modeled, in a theoretical sense, as a PRAM algorithm.

Beyond the low overhead discussed above, our compiled sequential O-NRAM has the additional benefit that it allows for an extremely simple prefetching algorithm. In recent work, Yu et al. [53] proposed a dynamic prefetching algorithm for ORAM, which greatly improved the practical performance of ORAM. We note that our parallel-to-sequential compiler achieves prefetching essentially for free: Since the underlying PRAM program will make many parallel memory accesses to each bank, and since the compiler knows these memory addresses ahead of time, these memory accesses can automatically be prefetched. We note that a similar observation was made by Vishkin [45], who suggested leveraging parallelism for performance improvement by using (compile-time) prefetching in serial or parallel systems.

"Parallel-to-Parallel" Compiler Finally, we consider oblivious simulation in the parallel setting. We show that for any parallel program executing in t parallel steps with $P = M^{1+\delta}$ processors, we can obliviously simulate the program on a Network PRAM with $P' := P'(\lambda) \in O(P/\log^* P)$ processors (where λ is security parameter), running in $O(t \log^* P)$ time, thereby achieving $O(\log^* P)$ blowup in parallel time and bandwidth, and optimal work. In comparison, the best-known OPRAM scheme has poly log N blowup in parallel time and bandwidth. The compiler works when the underlying program is in the EREW, CREW, common CRCW or arbitrary CRCW PRAM model. The resulting compiled program is in the arbitrary CRCW PRAM model. Therefore, this is the only result presented in this paper whose realization requires a PRAM-supporting architecture.

1.4. *Technical Highlights*

Our most interesting technique is for the parallel-to-parallel compiler. We achieve this through an intermediate stepping stone where we first construct a parallel-to-sequential compiler (which may be of independent interest).

At a high level, the idea is to assign each virtual address to a pseudorandom memory bank (and this assignment stays the same during the entire execution). Suppose that a program is sufficiently parallel such that it always makes memory requests in $P := P(\lambda) \in \omega(M \log N)$-sized batches. For now, assume that all memory requests within a batch operate on *distinct* virtual addresses—if not we can leverage a hash table to suppress duplicates, using an additional "scratch" bank as the CPU's working memory. Then, clearly each memory bank will in expectation serve P/M requests for each batch. With a simple Chernoff bound, we can conclude that each memory bank will serve $O(P/M)$ requests for each batch, except with *negligible* probability. In a sequential setting, we can easily achieve $O(1)$ bandwidth and runtime blowup: for each batch of memory requests, the CPU will sequentially access each bank $O(P/M)$ number of times, padding with dummy accesses if necessary (see Sect. 4).

However, additional difficulties arise when we try to execute the above algorithm in parallel. In each step, there is a batch of P memory requests, one coming from each processor. However, each processor cannot perform its own memory request, since the adversary can observe which processor is talking to which memory bank and can detect duplicates (note this problem did not exist in the sequential case since there was only one processor). Instead, we wish to

1. hash the memory requests into buckets according to their corresponding banks while suppressing duplicates; and
2. pad the number of accesses to each bank to a worst-case maximum—as mentioned earlier, if we suppressed duplicate addresses, each bank has $O(P/M)$ requests with probability $1 - \text{negl}(\lambda)$.

At this point, we can assign processors to the memory requests in a round-robin manner, such that which processor accesses which bank is "fixed". Now, to achieve the above two tasks in $O(\log^* P)$ parallel time, we need to employ non-trivial parallel algorithms for "colored compaction" [4] and "static hashing" [5,18], for the arbitrary CRCW PRAM model, while using a scratch bank as working memory (see Sect. 5).

1.5. *Related Work*

Oblivious RAM (ORAM) was first proposed in a seminal work by Goldreich and Ostrovsky [19,20] where they laid a vision of employing an ORAM-capable secure processor to protect software against piracy. In their work, Goldreich and Ostrovsky showed both a poly-logarithmic upper-bound (commonly referred to as the hierarchical ORAM framework) and a logarithmic lower-bound for ORAM—both under constant CPU cache. Goldreich and Ostrovsky's hierarchical construction was improved in several subsequent works [6,23,25,29,37,49–51]. Recently, Shi et al. proposed a new, tree-based paradigm for constructing ORAMs [41], thus leading to several new constructions that are simple and practically efficient [8,13,44,47]. Notably, circuit ORAM [47] partially

resolved the tightness of the Goldreich–Ostrovsky lower bound, by showing that certain stronger interpretations of their lower bound are indeed tight.

Theoretically, the best-known ORAM scheme (with constant CPU cache) for small $O(\log N)$-sized memory words[1] is a construction by Kushilevitz et al. [29], achieving $O(\log^2 N / \log \log N)$ bandwidth and runtime blowup. Path ORAM (variant with $O(1)$ CPU cache [48]) and circuit ORAM can achieve better bounds for bigger memory words. For example, circuit ORAM achieves $\widehat{O}(\log N)$ bandwidth blowup for a word size of $\Omega(\log^2 N)$ bits; and $\widehat{O}(\log N)$ runtime blowup for a memory word size of N^ϵ bits where $0 < \epsilon < 1$ is any constant within the specified range.

ORAMs with larger CPU cache sizes (caching up to N^α words for any constant $0 < \alpha < 1$) have been suggested for cloud storage outsourcing applications [23,43,51]. In this setting, Goodrich and Mitzenmacher [23] first showed how to achieve $O(\log N)$ bandwidth and runtime blowup.

Other than secure processors and cloud outsourcing, ORAM is also noted as a key primitive for scaling secure multiparty computation to big data [26,30,47,48]. In this context, Wang et al. [47,48] pointed out that the most relevant ORAM metric should be the circuit size rather than the traditionally considered bandwidth metrics. In the secure computation context, Lu and Ostrovsky [31] proposed a two-server ORAM scheme that achieves $O(\log N)$ runtime blowup. Similarly, ORAM can also be applied in other RAM model cryptographic primitives such as (reusable) Garbled RAM [14–16,32,33].

Goodrich and Mitzenmacher [23] and Williams et al. [52] observed that computational tasks with inherent parallelism can be transformed into efficient, oblivious counterparts in the traditional ORAM setting—but our techniques apply to the NRAM model of computation. Finally, oblivious RAM has been implemented in outsourced storage settings [42,43,49,51,52], on secure processors [9,11,12,34,35,39], and atop secure multiparty computation [26,47,48].

Comparison of Our Parallel-to-Parallel Compiler with the Work of [7] Recently, Boyle et al. [7] proposed oblivious parallel RAM, and presented a construction for oblivious simulation of PRAMs in the PRAM model. Our result is incomparable to their result: Our security model is weaker than theirs since we assume obliviousness within each memory bank comes for free; on the other hand, we obtain far better asymptotical and concrete performance. We next elaborate further on the differences in the results and techniques of the two works. Boyle et al. [7] provide a compiler from the EREW, CREW and CRCW PRAM models to the EREW PRAM model. The security notion achieved by their compiler provides security against adversaries who see the entire access pattern, as in standard oblivious RAM. However, their compiled program incurs a poly log overhead in both the parallel time and total work. Our compiler is a compiler from the EREW, CREW, common CRCW and arbitrary CRCW PRAM models to the arbitrary CRCW PRAM model and the security notion we achieve is the weaker notion of oblivious network RAM, which protects against adversaries who see the bank being accessed, but not the offset within the bank. On the other hand, our compiled program incurs only a $\log^*$ time overhead and its work is asymptotically the *same* as the underlying PRAM. Both our work and the work of [7] leverage previous results and techniques

[1] Every memory word must be large enough to store the logical memory address.

from the parallel computing literature. However, our techniques are primarily from the CRCW PRAM literature, while [7] use primarily techniques from the low-depth circuit literature, such as highly efficient sorting networks.

2. Definitions

2.1. *Background: Random Access Machines (RAM)*

We consider RAM programs to be interactive stateful systems $\langle \Pi, \mathsf{state}, D \rangle$, consisting of a memory array D of $N := N(\lambda)$ memory words, for polynomial $N(\lambda) \in \Omega(\lambda)$ and security parameter λ, a CPU state denoted state, and a next-instruction function Π which given the current CPU state and a value rdata read from memory, outputs the next instruction I and an updated CPU state denoted state':

$$(\mathsf{state}', I) \leftarrow \Pi(\mathsf{state}, \mathsf{rdata})$$

Each instruction I is of the form $I = (\mathsf{op}, \ldots)$, where op is called the op-code whose value is read, write, or stop. The initial CPU state is set to $(\mathsf{start}, *, \mathsf{state}_{\mathsf{init}})$. Upon input x, the RAM machine executes, computes output z and terminates. CPU state is reset to $(\mathsf{start}, *, \mathsf{state}_{\mathsf{init}})$ when the computation on the current input terminates.

On input x, the execution of the RAM proceeds as follows. If $\mathsf{state} = (\mathsf{start}, *, \mathsf{state}_{\mathsf{init}})$, set $\mathsf{state} := (\mathsf{start}, x, \mathsf{state}_{\mathsf{init}})$, and $\mathsf{rdata} := 0$. Now, repeat the $\mathsf{doNext}()$ till termination, where $\mathsf{doNext}()$ is defined as below:

$\mathsf{doNext}()$

1. Compute $(I, \mathsf{state}') = \Pi(\mathsf{state}, \mathsf{rdata})$. Set $\mathsf{state} := \mathsf{state}'$.
2. If $I = (\mathsf{stop}, z)$ then terminate with output z.
3. If $I = (\mathsf{write}, \mathsf{vaddr}, \mathsf{wdata})$ then set $D[\mathsf{vaddr}] := \mathsf{wdata}$.
4. If $I = (\mathsf{read}, \mathsf{vaddr}, \perp)$ then set $\mathsf{rdata} := D[\mathsf{vaddr}]$.

2.2. *Parallel RAM*

To formally characterize what it means for a program to exhibit a sufficient degree of parallelism, we will formally define a P-parallel RAM. In this section, the reader should think of parallelism as a property of the program to be simulated—we actually characterize costs assuming both the non-oblivious and the oblivious programs are executed on a sequential machine (different from Sect. 5).

An P-parallel RAM machine is the same as a RAM machine, except the next-instruction function outputs P instructions which can be executed in parallel.

Definition 1. (*P-parallel RAM*) An *P-Parallel RAM* is a RAM which has a next-instruction function $\Pi = \Pi_1, \ldots, \Pi_P$ such that on input $(\mathsf{state} = \mathsf{state}_1 || \cdots || \mathsf{state}_P, \mathsf{rdata} = \mathsf{rdata}_1 || \cdots || \mathsf{rdata}_P)$, Π outputs P instructions $(I_1, \ldots, I_P)$ and P updated

states $\mathsf{state}'_1, \ldots, \mathsf{state}'_P$ such that for $p \in [P]$, $(I_p, \mathsf{state}'_p) = \Pi_p(\mathsf{state}_p, \mathsf{rdata}_p)$. The instructions $I_1, \ldots, I_P$ satisfy one of the following:

- All of $I_1, \ldots, I_P$ are set to (stop, z) (with the same z).
- All of $I_1, \ldots, I_P$ are either of the form. $(\mathsf{read}, \mathsf{vaddr}, \perp)$ or $(\mathsf{write}, \mathsf{vaddr}, \mathsf{wdata})$.

Finally, the state state has size at most $O(P)$.

In an intermediate result in Sect. 4.1, we will consider a special case where in each parallel step of the PRAM execution, the memory requests made by each processor in the P-parallel RAM have distinct addresses—we refer to this model as a *restricted* PRAM. A formal definition follows.

Definition 2. (*Restricted P-parallel RAM*) For a P-parallel RAM denoted $\mathsf{PRAM} := \langle D, \mathsf{state}_1, \ldots, \mathsf{state}_P, \Pi_1, \ldots \Pi_P \rangle$, if every batch of instructions $I_1, \ldots, I_P$ have unique vaddr's, we say that PRAM is a *restricted* P-parallel RAM.

2.3. *Network RAM (NRAM)*

Network RAM A network RAM (NRAM) is the same as a regular RAM, except that memory is distributed across multiple banks, $\mathsf{Bank}_1, \ldots, \mathsf{Bank}_M$. In an NRAM, every virtual address vaddr can be written in the format $\mathsf{vaddr} := (m, \mathit{offset})$, where $m \in [M]$, for $M := M(k)$, is the bank identifier, and *offset* is the offset within the Bank_m.

Otherwise, the definition of NRAM is identical to the definition of RAM.

Probabilistic NRAM Similar to the probabilistic RAM notion formalized by Goldreich and Ostrovsky [19,20], we additionally define a *probabilistic NRAM*. A probabilistic NRAM is an NRAM whose CPU state is initialized with randomness ρ (that is unobservable to the adversary). If an NRAM is deterministic, we can simply assume that the CPU's initial randomness is fixed to $\rho := 0$. Therefore, a deterministic NRAM can be considered as a special case of a probabilistic NRAM.

Outcome of Execution Throughout the paper, we use the notation $\mathsf{RAM}(x)$ or $\mathsf{NRAM}(x)$ to denote the outcome of executing a RAM or NRAM on input x. Similarly, for a probabilistic NRAM, we use the notation $\mathsf{NRAM}_\rho(x)$ to denote the outcome of executing on input x, when the CPU's initial randomness is ρ.

2.4. *Oblivious Network RAM (O-NRAM)*

Observable Traces To define oblivious network RAM, we need to first specify which part of the memory trace an adversary is allowed to observe during a program's execution. As mentioned earlier in the introduction, each memory bank has trusted logic for encrypting and decrypting the memory offset. The offset within a bank is transferred in encrypted format on the memory bus. Hence, *for each memory access* $\mathsf{op} :=$ "read" *or* $\mathsf{op} :=$ "write" *to virtual address* $\mathsf{vaddr} := (m, \mathit{offset})$, *the adversary observes only the op-code* op *and the bank identifier* m, *but not the offset within the bank.*

Definition 3. (*Observable traces*) For a probabilistic NRAM, we use the notation $\mathsf{Tr}_\rho(\mathsf{NRAM}, x)$ to denote its observable traces upon input x, and initial CPU randomness ρ:

$$\mathsf{Tr}_\rho(\mathsf{NRAM}, x) := \left\{(\mathsf{op}_1, m_1), (\mathsf{op}_2, m_2), \ldots, (\mathsf{op}_T, m_T)\right\}$$

where T is the total execution time of the NRAM, and (op_i, m_i) is the op-code and memory bank identifier during step $i \in [T]$ of the execution.

We remark that one can consider a slight variant model where the op-codes $\{\mathsf{op}_i\}_{i\in[T]}$ are also hidden from the adversary. Since to hide whether the operation is a read or write, one can simply perform one read and one write for each operation—the differences between these two models are insignificant for technical purposes. Therefore, in this paper, we consider the model whose observable traces are defined in Definition 3).

Oblivious Network RAM Intuitively, an NRAM is said to be oblivious, if for any two inputs x_0 and x_1 resulting in the same execution time, their observable memory traces are computationally indistinguishable to an adversary.

For simplicity, we define obliviousness for NRAMs that run in deterministic T time regardless of the inputs and the CPU's initial randomness. One can also think of T as the worst-case runtime, and that the program is always padded to the worst-case execution time. Oblivious NRAM can also be similarly defined when its runtime is randomized—however we omit the definition in this paper.

Definition 4. (*Oblivious network RAM*) Consider an NRAM that runs in deterministic time $T := T(\lambda) \in \mathrm{poly}(\lambda)$. The NRAM is said to be computationally oblivious if no polynomial-time adversary $\mathcal{A}$ can win the following security game with more than $\frac{1}{2} + \mathrm{negl}(\lambda)$ probability. Similarly, the NRAM is said to be statistically oblivious if no adversary, even computationally unbounded ones, can win the following game with more than $\frac{1}{2} + \mathrm{negl}(\lambda)$ probability.

- $\mathcal{A}$ chooses two inputs x_0 and x_1 and submits them to a challenger.
- The challenger selects $\rho \in \{0,1\}^\lambda$, and a random bit $b \in \{0,1\}$. The challenger executes NRAM with initial randomness ρ and input x_b for exactly T steps, and gives the adversary $\mathsf{Tr}_\rho(\mathsf{NRAM}, x_b)$.
- $\mathcal{A}$ outputs a guess b' of b, and wins the game if $b' = b$.

2.5. *Notion of Simulation*

Definition 5. (*Simulation*) We say that a deterministic $\mathsf{RAM} := \langle \Pi, \mathsf{state}, D\rangle$ can be *correctly simulated* by another probabilistic $\mathsf{NRAM} := \langle \Pi', \mathsf{state}', D'\rangle$ if for any input x for any initial CPU randomness ρ, $\mathsf{RAM}(x) = \mathsf{NRAM}_\rho(x)$. Moreover, if NRAM is oblivious, we say that NRAM is an oblivious simulation of RAM.

Below, we explain some subtleties regarding the model, and define the metrics for oblivious simulation.

Uniform Versus Non-uniform Memory Word Size The O-NRAM simulation can either employ uniform memory word size or non-uniform memory word size. For example, the non-uniform word size model has been employed for recursion-based ORAMs in the literature [44,47]. In particular, Stefanov et al. describe a parametrization trick where they use a smaller word size for position map levels of the recursion [44].

Metrics for Simulation Overhead In the ORAM literature, several performance metrics have been considered. To avoid confusion, we now explicitly define two metrics that we will adopt later. If an NRAM correctly simulates a RAM, we can quantify the overhead of the NRAM using the following metrics.

- *Runtime Blowup* If a RAM runs in time T, and its oblivious simulation runs in time T', then the runtime blowup is defined to be T'/T. This notion is adopted by Goldreich and Ostrovsky in their original ORAM paper [19,20].
- *Bandwidth Blowup* If a RAM transfers Y bits between the CPU and memory, and its oblivious simulation transfers Y' bits, then the bandwidth blowup is defined to be Y'/Y. Clearly, if the oblivious simulation is in a uniform word size model, then bandwidth blowup is equivalent to runtime blowup. However, bandwidth blowup may not be equal to runtime blowup in a non-uniform word size model.

In this paper, we consider oblivious simulation of RAMs in the NRAM model, and we focus on the case when the oblivious NRAM has only $O(1)$ words of CPU cache.

2.6. *Network PRAM (NPRAM) Definitions*

Similar to our NRAM definition, an NPRAM is much the same as a standard PRAM, except that (1) memory is distributed across multiple banks, $\mathsf{Bank}_1, \ldots, \mathsf{Bank}_M$; and (2) every virtual address vaddr can be written in the format $\mathsf{vaddr} := (m, \mathit{offset})$, where m is the bank identifier, and *offset* is the offset within the Bank_m. We use the notation P-parallel NPRAM to denote an NPRAM with $P := P(\lambda)$ parallel processors, each with $O(1)$ words of cache. If processors are initialized with secret randomness unobservable to the adversary, we call this a probabilistic NPRAM.

Observable Traces In the NPRAM model, we assume that an adversary can observe the following parts of the memory trace: (1) which processor is making the request; (2) whether this is a read or write request; and (3) which bank the request is going to. The adversary is unable to observe the offset within a memory bank.

Definition 6. (*Observable traces for NPRAM*) For a probabilistic P-parallel NPRAM, we use $\mathsf{Tr}_\rho(\mathsf{NPRAM}, x)$ to denote its observable traces upon input x, and initial CPU randomness ρ (collective randomness over all processors):

$$\mathsf{Tr}_\rho(\mathsf{NPRAM}, x) := \left[\left(\left(\mathsf{op}_1^1, m_1^1\right), \ldots, \left(\mathsf{op}_1^P, m_1^P\right)\right), \ldots, \left(\left(\mathsf{op}_T^1, m_T^1\right), \ldots, \left(\mathsf{op}_T^P, m_T^P\right)\right)\right]$$

where T is the total parallel execution time of the NPRAM, and $\{(\mathsf{op}_i^1, m_i^1), \ldots, (\mathsf{op}_i^P, m_i^P)\}$ is of the op-codes and memory bank identifiers for each processor during parallel step $i \in [T]$ of the execution.

Based on the above notion of observable memory trace, an oblivious NPRAM can be defined in a similar manner as the notion of O-NRAM (Definition 4).

Metrics We consider classical metrics adopted in the vast literature on parallel algorithms, namely, the parallel runtime and the total work. In particular, to characterize the oblivious simulation overhead, we will consider

- *Parallel Runtime Blowup* The blowup of the parallel runtime comparing the O-NPRAM and the NPRAM.
- *Total Work Blowup* The blowup of the total work comparing the O-NPRAM and the NPRAM. If the total work blowup is $O(1)$, we say that the O-NPRAM achieves *optimal* total work.

3. Sequential Oblivious Simulation

3.1. *First Attempt: Oblivious NRAM with $O(M)$ CPU Cache*

Let $M := M(\lambda)$ denote the number of memory banks in our NRAM, where each bank has $O(N/M)$ capacity. We first describe a simple oblivious NRAM with $O(M)$ CPU private cache. Under a non-uniform memory word size model, Our O-NRAM construction achieves $O(1)$ bandwidth blowup under $\Omega(\log^2 N)$ memory word size. Later, in Sect. 3.2, we will describe how to reduce the CPU cache to $O(1)$ by introducing an additional scratch memory bank of $O(M)$ in size. In particular, an interesting parametrization point is when $M \in O(\sqrt{N})$.

Our idea is inspired by the partition-based ORAM idea described by Stefanov, Shi, and Song [43]. For simplicity, like in many earlier ORAM works [20,41], we focus on presenting the algorithm for making memory accesses, namely the Access algorithm. A description of the full O-NRAM construction is apparent from the Access algorithm: basically, the CPU interleaves computation (namely, computing the next-instruction function Π) with the memory accesses.

CPU Private Cache The CPU needs to maintain the following metadata:

- A *position map* that stores which bank each memory word currently resides in. We use the notation position[vaddr] to denote the bank identifier for the memory word at virtual address vaddr. Although storing the position map takes $O(N \log M)$ bits of CPU cache, we will later describe a recursion technique [41,43] that can reduce this storage to $O(1)$; and
- An *eviction cache* consisting of M *queues*. The queues are used for temporarily buffering memory words before they are obliviously written back to the memory banks. Each queue $m \in [M]$ can be considered as an extension of the mth memory bank. The eviction cache is $O(M) + f(N)$, for any $f(N) = \omega(\log N)$ in size. For now, consider that the eviction cache is stored in the CPU cache, such that accesses to the eviction cache do not introduce memory accesses. Later in Sect. 3.2, we will move the eviction cache to a separate scratch bank—it turns out there is a small technicality with that requiring us to use a deamortized Cuckoo hash table [2].

Memory Access Operations Figure 1 describes the algorithm for making a memory access. To access a memory word identified by virtual address vaddr, the CPU first looks up the position map $m :=$ position[vaddr] to find the bank identifier m where the memory word vaddr currently resides. Then, the CPU fetches the memory word vaddr

```
Access(op, vaddr, wdata):
 1: m̃ ← UniformRandom(1 . . . M)
 2: m := position[vaddr], position[vaddr] := m̃
 3: if word at vaddr is in queue[m] then
 4:     rdata := queue[m].ReadAndRm(vaddr)
 5:     ReadBank(m, ⊥)
 6: else
 7:     queue[m].ReadAndRm(⊥)
    /* Dummy operation, needed when the eviction queues are stored in a scratch bank, see Section 3.2 */
 8:     rdata := ReadBank(m, vaddr)
    /* Each bank implements a hash table with good worst-case cost (Theorem 1). */
 9: end if
10: if op = read then wdata := rdata
11: queue[m̃] := queue[m̃].push(vaddr, wdata)
12: Call SeqEvict(ν)
13: return rdata
```

Fig. 1. Algorithm for data access. Read or write a memory word identified by vaddr. If op = read, the input parameter wdata = None, and the Access operation returns the newly fetched word. If op = write, the Access operation writes the specified wdata to the memory word identified by vaddr, and returns the old value of the word at vaddr.

from either the queue queue[m] or the bank m. In the former case, the ReadAndRm primitive sequentially scans through each element in queue[m] to retrieve data stored at vaddr. For the latter case, since the set of vaddr's stored in any bank may be discontinuous, we first assume that each bank implements a hash table such that one can look up each memory location by its vaddr using ReadBank. Later, we will describe how to instantiate this hash table (Theorem 1).

After fetching the memory word vaddr, the CPU assigns it to a fresh random bank $\widetilde{m}$. However, to avoid leaking information, the memory word is not immediately written back to the bank $\widetilde{m}$. Instead, recall that the CPU maintains M queues for buffering memory words before write-back. At this point, the memory word at address vaddr is added to queue[$\widetilde{m}$]—signifying that the memory word vaddr is scheduled to be written back to $\mathsf{Bank}_{\widetilde{m}}$.

Background Eviction To prevent the CPU's eviction queues from overflowing, a background eviction process obliviously evicts words from the queues back to the memory banks. One possible eviction strategy is that, on each data access, the CPU chooses $\nu = 2$ queues for eviction—by sequentially cycling through the queues. When a queue is chosen for eviction, an arbitrary word is popped from the queue and written back to the corresponding memory bank. If the chosen queue is empty, a dummy word is evicted to prevent leaking information. Stefanov et al. proved that such an eviction process is fast enough so that the CPU's eviction cache load is bounded by $O(M)$ except with negligible probability [43]—assuming that $M \in \omega(\log N)$.

Lemma 1. *The CPU's eviction cache, i.e., the total capacity of all eviction queues, is bounded by* $O(M) + f(N)$*, for any* $f(N) = \omega(\log N)$ *words except with* $\mathrm{negl}(\lambda)$ *probability.*

```
Evict(m):
1: if len(queue[m]) = 0 then
2:     WriteBank(m, ⊥, None)
3: else
4:     (vaddr, wdata) := queue[m].pop()
5:     WriteBank(m, vaddr, wdata)
6: end if
```

```
SeqEvict(ν):
// Let cnt denote a stateful counter.
1: Repeat ν times:
2:     cnt := (cnt + 1) mod M
3:     Evict(cnt)
```

Fig. 2. Background eviction algorithms with eviction rate ν. SeqEvict linearly cycles through the eviction queues to evict from. If a queue selected for eviction is empty, evict a dummy word for obliviousness. Counter cnt is a global variable.

Proof. The proof follows from Stefanov et al. [43], and is a straightforward application of Chernoff bound. □

Instantiating the Per-Bank Hash Table In Figs. 1 and 2, we assume that each bank implements a hash table with good worst-case performance. We now describe how to instantiate this hash table to achieve $\widehat{O}(1)$ cost per operation except with negligible failure probability.

A first idea is to implement a standard Cuckoo hash table [38] for each memory bank. In this way, lookup is worst-case constant time, whereas insertion is average-case constant time, and worst-case $\widehat{O}(\log N)$ time to achieve a failure probability negligible in N. To ensure obliviousness, we can not reveal the insertion time—for example, insertion time can reveal the current usage of each bank, which in turns leaks additional information about the access pattern. However, we do not wish to incur this worst-case insertion time upon every write-back.

To deal with this issue, we will rely on a deamortized Cuckoo hash table such as the one described by Arbitman et al. [2]. Their idea is to rely on a small queue that temporarily buffers the pending work associated with insertions. Upon every operation, perform a fixed amount of work at a rate faster than the amortized insertion cost of a standard Cuckoo hash table. For our application, we require that the failure probability be negligible in security parameter, λ. Therefore, we introduce a modified version of Arbitman et al.'s theorem [2] as stated below.

Theorem 1. (Deamortized Cuckoo hash table: negligible failure probability version) *There exists a deamortized Cuckoo hash table of capacity $s := s(\lambda) \in \text{poly}(N)$ such that with probability $1 - \text{negl}(\lambda)$, each insertion, deletion, and lookup operation is performed in worst-case $\widehat{O}(1)$ time (not counting the cost of operating on the queue)—as long as at any point in time at most s elements are stored in the data structure. The above deamortized Cuckoo hash table consumes $O(s) + O(N^{\delta})$ space where $0 < \delta < 1$ is a constant.*

In the above, the $O(s)$ part of the space corresponds to the two tables T_0 and T_1 for storing the elements of the hash table, and the additional $O(N^{\delta})$ space is for implementing the pending work queue (see Arbitman et al. [2] for more details). Specifically, Arbitman et al. suggested that the work queue be implemented with constant number k

of standard hash tables which are N^{δ}, for $\delta < 1$, in size. To achieve negligible failure probability, we instead set $k = k(N)$ to be any $k(N) \in \omega(1)$. We will discuss the details of our modified construction and analysis in Sect. 6.

Recursion In the above scheme, the CPU stores both a position map of $\Theta(N \log N)$ bits, and an eviction cache containing $\Theta(M)$ memory words. On each data access, the CPU reads $\Theta(w)$ bits assuming each memory word is of w bits. Therefore, the bandwidth blowup is $O(1)$.

We now show how to rely on a recursion idea to avoid storing this position map inside the CPU—for now, assume that the CPU still stores the eviction cache, which we will get rid of in Sect. 3.2. The idea is to recursively store the position map in smaller oblivious NRAMs.

In particular, consider ONRAM_0 to be the actual data ONRAM, whose position map contains N words. Then, the position map can be organized into N/c *blocks* by combining each c words into one data block. In this case, the position map can be stored in a ONRAM, called ONRAM_1, of capacity N/c with a block size to be cw. Since ONRAM_1 also needs to store a position map, we can recursively apply the same idea to construct $\mathsf{ONRAM}_2, \mathsf{ONRAM}_3, \ldots,$. Note that the capacity for ONRAM_k is N/c^k, while its word size is always cw. Given c is a constant, e.g., $c \geq 2$, we know that there are at most $k = \log N / \log c \in O(\log N)$ level of recursions to reach ONRAM_k with a constant capacity. In this structure, ONRAM_0 is the actual data ONRAM, while others are metadata ONRAMs.

To access a memory word in ONRAM_0, the client first makes a position map lookup in ONRAM_1 which triggers a recursive call to look up the position of the position in ONRAM_2, and so on. The original binary-tree ORAM [41] described a simple way to parametrize the recursion, using a *uniform* memory word size across all recursion levels. Later schemes [44,47], however, described new tricks to parametrize the recursion, where a different memory word size is chosen for all the metadata levels than the data level (i.e., ONRAM_0)—the latter trick allows one to reduce the bandwidth blowup for reasonably big memory words size. Below, we describe these parametrizations, state the bandwidth blowup and runtime blowup we achieve in each setting. Recall that as mentioned earlier, the bandwidth blowup and runtime blowup equate for a uniform memory word size setting; however, under non-uniform memory word sizes, the two metrics may not equate.

- *Uniform Memory Word Size* The depth of recursion is smaller when the memory word is larger.
  - Assume that each memory word is at least $c \log N$ bits in size for some constant $c > 1$. In this case, the recursion depth is $O(\log N)$. Hence, the resulting O-NRAM has $\widehat{O}(\log N)$ runtime blowup and bandwidth blowup.
  - Assume that each memory word is at least N^{ϵ} bits in size for some constant $0 < \epsilon < 1$. In this case, the recursion depth is $O(1)$. Hence, the resulting O-NRAM has $\widehat{O}(1)$ runtime blowup and bandwidth blowup.

- *Non-uniform Memory Word Size* Using a parametrization trick by Stefanov et al. [44], we can parametrize the position map recursion levels to have a dif-

ferent word size than the data level. Of particular interest is the following point of parametrization:

– Assume that each memory word of the original RAM is $W := W(\lambda) \in \Omega(\log^2 N)$ bits—this will be the word size for the data level of the recursion. For the position map levels, we will use a word size of $\Theta(\log N)$ bits. In this way, the recursion depth is $O(\log N)$. For each access, the total number of bits transferred include one data word of W bits, and $O(\log N)$ words of $O(\log N)$ bits. Thus, we achieve $\widehat{O}(1)$ bandwidth blowup, but $\widehat{O}(\log N)$ runtime blowup.

Finally, observe that *we need not create separate memory banks for each level of the recursion*. In fact, the recursion levels can simply reuse the memory banks of the top data level, introducing only a constant factor blowup in the size of the memory bank.

3.2. *Achieving $O(1)$ Words of CPU Cache*

We now explain how to reduce the CPU cache size to $O(1)$, while storing the eviction queues in a separate scratch bank. It turns out that there is a small technicality when we try to do so, requiring the use of a special data structure as described below. When we move the eviction queues to the scratch bank, we would like each queue to support the operations: pop(), push() and ReadAndRm(), as required by algorithms in Figs. 1 and 2 with worst-case $\widehat{O}(1)$ cost except with negl(λ) failure probability. While a simple queue supports pop() and push() with these time bounds, it does not support ReadAndRm(). To achieve this, the scratch bank will maintain the following structures:

- Store M eviction queues supporting only pop() and push() operations. The total number of elements in all queues does not exceed $O(M) + f(N)$ for any $f(N) \in \omega(\log N)$ except with negligible failure probability. It is not hard to see that these M eviction queues can be implemented with $O(M) + f(N)$ for any $f(N) \in \omega(\log N)$ space in total and $O(1)$ cost per operation.
- Separately, store the entries of all M eviction queues in a single deamortized Cuckoo hash table [2] inside the scratch bank. Such a deamortized Cuckoo hash table can achieve $\widehat{O}(1)$ cost per operation (insertion, removal, lookup) except with negligible failure probability. When an element is *popped from* or *pushed to* any of the eviction queues, it is also inserted or removed in this big deamortized Cuckoo hash table. However, when an element must be *read and removed* from any of the eviction queues, then the element is looked up from the big hash table and it is just marked as *deleted*. When time comes for this element to be popped from some queue during the eviction process, a dummy eviction is performed.

Theorem 2. (O-NRAM simulation of arbitrary RAM programs: *uniform* word size model) *Any N-word RAM with a word size of $W = f(N) \log N$ bits can be simulated by an oblivious NRAM that consumes $O(1)$ words of CPU cache, and with $O(M)$ memory banks each of $O(M + N/M + N^{\delta})$ words in size, for any constant $0 < \delta < 1$. The oblivious NRAM simulation incurs $\widehat{O}(\log_{f(N)} N)$ runtime blowup and bandwidth blowup. As special cases of interest:*

- *When the word size is $W = N^{\epsilon}$ bits, the runtime blowup and bandwidth blowup are both $\widehat{O}(1)$.*

- *When the word size is* $W = c \log N$ *bits for some constant* $c > 1$*, the runtime blowup and bandwidth blowup are both* $\widehat{O}(\log N)$.

Theorem 3. (O-NRAM simulation of arbitrary RAM programs: *non-uniform* word size model) *Any* N*-word RAM with a word size of* $W = \Omega(\log^2 N)$ *bits can be simulated by an oblivious NRAM (with non-uniform word sizes) that consumes* $O(W)$ *bits of CPU cache, and with* $O(M)$ *memory banks each of* $O(W \cdot (M + N/M + N^{\delta}))$ *bits in size. Further, the oblivious NRAM simulation incurs* $\widehat{O}(1)$ *bandwidth blowup and* $\widehat{O}(\log N)$ *runtime blowup.*

Note that for the non-uniform version of the theorem, we state the memory bank and cache sizes in terms of *bits* instead of *words* to avoid confusion. In both the uniform and non-uniform versions of the theorem, an interesting point of parametrization is when $M = O(\sqrt{N})$, and each bank is $O(W\sqrt{N})$ *bits* in size. The proofs for these two theorems follow directly the analysis for the recursive construction of oblivious NRAM from Sect. 3.1, given each access to the scratch bank costs only a constant overhead.

4. Sequential Oblivious Simulation of Parallel Programs

We are eventually interested in parallel oblivious simulation of parallel programs (Sect. 5). As a stepping stone, we first consider sequential oblivious simulation of parallel programs. However, we emphasize that the results in this section can be of independent interest. In particular, one way to interpret these results is that "parallelism facilitates obliviousness". Specifically, if a program exhibits a sufficient degree of parallelism, then this program can be made oblivious at only const overhead in the network RAM model. The intuition for why this is so, is that instructions in each parallel time step can be executed in any order. Since subsequences of instructions can be executed in an arbitrary order during the simulation, many sequences of memory requests can be mapped to the same access pattern, and thus the request sequence is partially obfuscated.

4.1. *Warmup: Restricted Parallel RAM to Oblivious NRAM*

Our goal is to compile any P-parallel RAM (not necessarily restricted), into an efficient O-NRAM. As an intermediate step that facilitates presentation, we begin with a basic construction of O-NRAM from any *restricted*, parallel RAM. In the following section, we extend to a construction of O-NRAM from any parallel RAM (not necessarily restricted). Since we present our construction for the most general case—when the underlying PRAM is in the CRCW PRAM model—it follows that our final compiler works when the underlying P-parallel RAM is in the EREW, CREW, or common/arbitrary/priority CRCW PRAM model.

Let $\mathsf{PRAM} := \langle D, \mathsf{state}_1, \ldots, \mathsf{state}_P, \Pi_1, \ldots \Pi_P \rangle$ be a restricted P-Parallel RAM, for $P := P(\lambda) \in \omega(M \log N)$. We now present an O-NRAM simulation of PRAM that requires $M + 1$ memory banks, each with $O(N/M + P)$ physical memory, where N is the database size.

doNext(): *//We only consider* read *and* write *instructions here but not* stop.

1: **For** $p := 1$ to P: $(\mathsf{op}_p, \mathsf{vaddr}_p, \mathsf{wdata}_p) := \Pi_p(\mathsf{state}_p, \mathsf{rdata}_p)$

2: $(\mathsf{rdata}_1, \mathsf{rdata}_2, \ldots, \mathsf{rdata}_p) := \mathsf{Access}\left(\{\mathsf{op}_p, \mathsf{vaddr}_p, \mathsf{wdata}_p\}_{p\in[P]}\right)$

Fig. 3. Oblivious simulation of each step of the restricted parallel RAM.

Setup: Pseudorandomly Assign Memory Words to Banks[2] The setup phase takes the initial states of the PRAM, including the memory array D and the initial CPU state, and compiles them into the initial states of the oblivious NRAM denoted ONRAM.

To do this, the setup algorithm chooses a secret key K, and sets ONRAM.state $=$ PRAM.state$||K$. Each memory bank of ONRAM will be initialized as a Cuckoo hash table. Each memory word in the PRAM's initial memory array D will be inserted into the bank numbered $(\mathsf{PRF}_K(\mathsf{vaddr}) \mod M) + 1$, where vaddr is the virtual address of the word in PRAM. Note that the ONRAM's $(M + 1)$th memory bank is reserved as a scratch bank whose usage will become clear later.

Simulating Each Step of the PRAM's Execution Each doNext() operation of the PRAM will be compiled into a sequence of instructions of the ONRAM. We now describe how this compilation works. Our presentation focuses on the case when the next instruction's op-codes are reads or writes. Wait or stop instructions are left unmodified during the compilation.

As shown in Fig. 3, for each doNext instruction, we first compute the batch of instructions $I_1, \ldots, I_P$, by evaluating the P parallel next-instruction circuits $\Pi_1, \ldots, \Pi_P$. This results in P parallel read or write memory operations. This batch of P memory operations (whose memory addresses are guaranteed to be distinct in the restricted parallel RAM model) will then be served using the subroutine Access.

We now elaborate on the Access subroutine. Each batch will have $P := P(\lambda) \in \omega(M \log N)$ memory operations whose virtual addresses are distinct. Since each virtual address is randomly assigned to one of the M banks, in expectation, each bank will get $P/M = \omega(\log N)$ hits. Using a balls-and-bins analysis, we show that the number of hits for each bank is highly concentrated around the expectation. In fact, the probability of any constant factor, multiplicative deviation from the expectation is negligible in N (and therefore also negligible in λ, since $N \geq \lambda$). Therefore, we choose $\mathsf{max} := 2(P/M)$ for each bank, and make precisely max number of accesses to each memory bank. Specifically, the Access algorithm first scans through the batch of $P \in \omega(M \log N)$ memory operations, and assigns them to M queues, where the mth queue stores requests assigned to the mth memory bank. Then, the Access algorithm sequentially serves the

[2] In fact, it is possible to use here a k-wise independent hash function, instead of a PRF, as long as k is sufficiently large. In particular, we must choose k so that the k-wise independent hash function "fools" Chernoff bounds. As shown in the seminal work of [40], this can be achieved by setting $k := k(N)$ for any function such that $k(N) \in \omega(\log N)$. In fact, "almost" k-wise independent hash functions [1] can also be used for load-balancing. Leveraging the analysis from [36], it can be shown that such hash function can be constructed such that every invocation is poly$(\log \log N)$ cost per evaluation. In this paper, our model assumes that the cost to evaluate the hash is a unit cost.

```
Access ({op_p, vaddr_p, wdata_p}_{p∈P}):
1: for p = 1 to P do
2:     m ← (PRF_K(vaddr_p) mod M) + 1;
3:     queue[m] := queue[m].push(p, op_p, vaddr_p, wdata_p);
        // queue is stored in a separate scratch bank.
4: end for
5: for m = 1 to M do
6:     if |queue[m]| > max then abort
7:     Pad queue[m] with dummy entries (⊥, ⊥, ⊥, ⊥) so that its size is max;
8:     for i = 1 to max do
9:         (p, op, vaddr, wdata) := queue[m].pop()
10:        rdata_p := ReadBank(m, vaddr)
       // Each bank is a deamortized Cuckoo hash table.
11:        if op = read then wdata := rdata_p
12:        WriteBank(m, vaddr, wdata)
13:    end for
14: end for
15: return (rdata_1, rdata_2, ..., rdata_P)
```

Fig. 4. Obliviously serving a batch of P memory requests with distinct virtual addresses.

requests to memory banks $1, 2, \ldots, M$, padding the number of accesses to each bank to max. This way, the access patterns to the banks are guaranteed to be oblivious.

The description of Fig. 4 makes use of M queues with a total size of $P \in \omega(M \log N)$ words. It is not hard to see that these queues can be stored in an additional scratch bank of size $O(P)$, incurring only constant number of accesses to the scratch bank per queue operation. Further, in Fig. 4, the time at which the queues are accessed, and the number of times they are accessed are not dependent on input data (notice that Line 7 can be done by linearly scanning through each queue, incurring a max cost each queue).

Cost Analysis Since $\mathsf{max} = 2(P/M)$, in Fig. 4 (see Theorem 4), it is not hard to see each batch of $P \in \omega(M \log N)$ memory operations will incur $\Theta(P)$ accesses to data banks in total, and $\Theta(P)$ accesses to the scratch bank. Therefore, the ONRAM incurs only a constant factor more total work and bandwidth than the underlying PRAM.

Theorem 4. *Let* PRF *be a family of pseudorandom functions,* PRAM *be a restricted P-Parallel RAM for $P := P(\lambda) \in \omega(M \log N)$, and let* $\mathsf{max} := 2(P/M)$. *Then, the construction described above is an oblivious simulation of* PRAM *using M banks each of size $O(N/M + P)$ words. The oblivious simulation performs total work that is constant factor larger than that of the underlying* PRAM.

Proof. Assuming the execution never aborts (Line 6 in Fig. 4), then Theorem 4 follows immediately, since the access pattern is deterministic and independent of the inputs. Therefore, it suffices to show that the abort happens with negligible probability on Line 6. This is shown in the following lemma. □

Lemma 2. *Let* $\mathsf{max} := 2(P/M)$. *For any* PRAM *and any input x, abort on Line 6 of Fig. 4 occurs only with negligible probability (over choice of the* PRF*).*

Remark 1. We note that with our choice of parameters, $P := P(\lambda) \in \omega(M \log N)$, the abort probability is at most $M \cdot \exp(-\frac{P}{3M})$. Since we assume $N \geq \lambda$ and $N \in \text{poly}(\lambda)$, we can choose a particular $f(\lambda) \in \omega(\log(\lambda))$ and set $P := M \cdot f(\lambda)$, thus achieving abort probability at most $M \cdot \exp(-f(\lambda)/3)$. As a concrete instantiation, we can set $\lambda = 2^{12} = 4096$ and $f(\lambda) := \log^2(\lambda)$, achieving abort probability $\exp(-48) < 2^{-70}$, corresponding to more than 70 bits of security. Since in practice, the size of memory, N, will be far larger than 4096 words, we believe the above settings are reasonable.

Proof. We first replace PRF with a truly random function f. Note that if we can prove the lemma for a truly random function, then the same should hold for PRF, since otherwise we obtain an adversary breaking pseudorandomness.

We argue that the probability that abort occurs on Line 6 of Fig. 4 in a particular step i of the execution is negligible. By taking a union bound over the (polynomial number of) steps of the execution, the lemma follows.

To upper bound the probability of abort in some step i, consider a thought experiment where we change the order of sampling the random variables: We run $\mathsf{PRAM}(x)$ to precompute all the PRAM's instructions up to and including the ith step of the execution (independently of f), obtaining P distinct virtual addresses, and only then choose the outputs of the random function f on the fly. That is, when each virtual memory address vaddr_p in step i is serviced, we choose $m := f(\mathsf{vaddr}_p)$ uniformly and independently at random. Thus, in step i of the execution, there are P distinct virtual addresses (i.e., balls) to be thrown into M memory banks (i.e., bins). For $P \in \omega(M \log N)$, we have expected load $P/M \in \omega(\log N)$ and so the probability that there exists a bin $i \in M$ whose load, load_i, exceeds $2(P/M)$ is

$$
\begin{aligned}
\Pr[\mathsf{load}_i > 2(P/M) \text{ for some } i \in [M]] &\leq \sum_{i \in M} \Pr[\mathsf{load}_i > 2(P/M)] \\
&= \sum_{i \in M} \Pr[\mathsf{load}_i > (1+1)(1/M \cdot P)] \\
&\leq M \cdot \exp\left(-\frac{P}{3M}\right) \qquad (4.1) \\
&\leq M \cdot N^{-\omega(1)} \qquad (4.2) \\
&= \text{negl}(N) \\
&= \text{negl}(\lambda), \qquad (4.3)
\end{aligned}
$$

where (4.1) follows due to standard multiplicative Chernoff bounds and (4.2) follows since $P/M = \omega(\log N)$. □

We note that in order for the above argument to hold, the input x cannot be chosen adaptively, and must be fixed before the PRAM emulation begins.

Access $(\{\mathsf{op}_p, \mathsf{vaddr}_p, \mathsf{wdata}_p, p\}_{p\in P})$:

/* *The* HTable *and* queue *data structures are stored in a scratch bank. Each entry of the hash table* $\mathsf{HTable}[\mathsf{op}_p, \mathsf{vaddr}_p]$ *consists of a* 3*-element array with indeces* $\{0,1,2\}$*. For obliviousness, operations on these data structures must be padded to the worst-case cost as we elaborate in the text.*/

1: **for** $p = 1$ to P: $\mathsf{HTable}[\mathsf{op}_p, \mathsf{vaddr}_p] := (\mathsf{wdata}_p, \perp, p)$ // *hash table insertions;*
2: **for** $\{(\mathsf{op}, \mathsf{vaddr}), \mathsf{wdata}, p\} \in \mathsf{HTable}$ **do** // *iterate through hash table*
3: $m := (\mathsf{PRF}_K(\mathsf{vaddr}) \mod M) + 1$
4: $\mathsf{queue}[m] := \mathsf{queue}[m].\mathsf{push}(\mathsf{op}, \mathsf{vaddr}, \mathsf{wdata})$;
5: **end for**
6: **for** $m = 1$ to M **do**
7: **if** $|\mathsf{queue}[m]| > \mathsf{max}$ **then abort**
8: Pad $\mathsf{queue}[m]$ with dummy entries $(\perp, \perp, \perp)$ so that its size is max;
9: **for** $i = 1$ to max **do**
10: $(\mathsf{op}, \mathsf{vaddr}, \mathsf{wdata}, p) := \mathsf{queue}[m].\mathsf{pop}()$
11: $\mathsf{rdata} := \mathsf{ReadBank}(m, \mathsf{vaddr})$
12: **if** op = read **then** wdata := rdata
13: $\mathsf{WriteBank}(m, \mathsf{vaddr}, \mathsf{wdata})$
14: $\mathsf{HTable}[\mathsf{op}, \mathsf{vaddr}] := (\mathsf{wdata}, \mathsf{rdata}, p)$ // *hash table updates*
15: **end for**
16: **end for**
17: **return** $(\mathsf{HTable}[\mathsf{op}_1, \mathsf{vaddr}_1][1], \ldots, \mathsf{HTable}[\mathsf{op}_P, \mathsf{vaddr}_P][1])$ // *hash table lookups*

Fig. 5. Obliviously serving a batch of P *memory request, not necessarily with distinct virtual addresses.* The current description allows for the underlying PRAM to be EREW, CREW, common/arbitrary/priority CRCW, where we assume priority CRCW gives priority to maximum processor id p (priority for minimum processor id can be supported by iterating from $p = P$ to 1 in line 1).

4.2. *Parallel RAM to Oblivious NRAM*

Use a Hash Table to Suppress Duplicates In Sect. 4.1, we describe how to obliviously simulate a restricted parallel RAM in the NRAM model. We now generalize this result to support any P-parallel RAM, not necessarily restricted ones. The difference is that for a generic P-parallel RAM, each batch of P memory operations generated by the next-instruction circuit need not have distinct virtual addresses. For simplicity, imagine that the entire batch of memory operations are reads. In the extreme case, if all $P \in \omega(M \log N)$ operations correspond to the same virtual address residing in bank m, then the CPU should not read bank m as many as P number of times. To address this issue, we rely on an additional Cuckoo hash table [38] denoted HTable to suppress the duplicate requests (see Fig. 5, and the doNext function is defined the same way as Sect. 4.1).

The HTable will be stored in the scratch bank. For simplicity of presentation, we employ a fully deamortized Cuckoo hash table [21,22].[3] As shown in Fig. 5, we need to support hash table insertions, lookups, and moreover, we need to be able to iterate through the hash table. We now make a few remarks important for ensuring obliviousness. Line 1 of Fig. 5 performs $P \in \omega(M \log N)$ number of insertions into the Cuckoo hash table. Due to the analysis of [21,22], we know that these insertions will take $O(P)$ number

[3] Our setting does not require all properties of a fully deamortized Cuckoo hash table. First, we require only *static* hashing and second, since we execute all insertions/lookups in batches of size P, we require only that any batch of P insertions/lookups (for an arbitrary set of keys that may include duplicates), takes time $O(P)$ with all but negligible probability.

of accesses with all but negligible probability. Therefore, to execute Line 1 obliviously, we simply need to pad with dummy memory accesses to the scratch bank up to some $\mathsf{max}' = c \cdot P$, for an appropriate constant c.

Next, we describe how to execute the loop at Line 2 obliviously. The total size of the Cuckoo hash table is $O(P)$. To iterate over the hash table, we simply make a linear scan through the hash table. Some entries will correspond to dummy elements. When iterating over these dummy elements, we simply perform dummy operations for the *for* loop. Finally, observe that Line 17 performs a batch of P lookups to the Cuckoo hash table. Again, due to the analysis of [21,22], we know that these lookups will take $O(P)$ number of accesses to the scratch bank with all but negligible probability.

Cost Analysis Since $\mathsf{max} = 2(P/M)$ (see Theorem 4), it is not hard to see each batch of $P = \omega(M \log N)$ memory operations will incur $O(P)$ accesses to data banks in total, and $O(P)$ accesses to the scratch bank. Note that this takes into account the fact that Line 1 and the for-loop starting at Line 2 are padded with dummy accesses. Therefore, the ONRAM incurs only a constant factor more total work and bandwidth than the underlying PRAM.

Theorem 5. *Let* $\mathsf{max} = 2(P/M)$. *Assume that* PRF *is a secure pseudorandom function, and* PRAM *is a* P*-Parallel RAM for* $P := P(\lambda) \in \omega(M \log N)$. *Then, the above construction obliviously simulates* PRAM *in the NRAM model, incurring only a constant factor blowup in total work and bandwidth consumption.*

Proof. (*Sketch*) Similar to the proof of Theorem 4, except that now we have the additional hash table. Note that obliviousness still holds, since, as discussed above, each batch of P memory requests requires $O(P)$ accesses to the scratch bank, and this can be padded with dummy accesses to ensure the number of scratch bank accesses remains the same in each execution. □

5. Parallel Oblivious Simulation of Parallel Programs

In the previous section, we considered sequential oblivious simulation of programs that exhibit parallelism—there, we considered parallelism as being a property of the program which will actually be executed on a sequential machine. In this section, we consider *parallel* and oblivious simulations of parallel programs. Here, the programs will actually be executed on a parallel machine, and we consider classical metrics such as parallel runtime and total work as in the parallel algorithms literature.

We introduce the *Network PRAM* model—informally, this is a network RAM with parallel processing capability (see Sect. 2.6 for the formal definitions). Our goal in this section will be to compile a PRAM into an oblivious network PRAM (O-NPRAM), *a.k.a.,* the "parallel-to-parallel compiler".

Our O-NPRAM is the network RAM analog of the oblivious parallel RAM (OPRAM) model by Boyle et al. [7] (see Sect. 2.6 for the formal definitions). Goldreich and Ostrovsky's logarithmic ORAM lower bound (in the sequential execution model) directly implies the following lower bound for standard OPRAM [7]: Let PRAM be an arbitrary PRAM with P processors running in parallel time t. Then, any P-parallel OPRAM sim-

ulating PRAM must incur $\Omega(t \log N)$ parallel time. Clearly, OPRAM would also work in our network RAM model albeit not the most efficient, since it is not exploiting the fact that the addresses in each bank are inherently oblivious. In this section, we show how to perform oblivious parallel simulation of "sufficiently parallel" programs in the network RAM model, incurring only $O(\log^* N)$ blowup in parallel runtime, and achieving optimal total work. Our techniques make use of fascinating results in the parallel algorithms literature [4,5,27].

5.1. *Construction of Oblivious Network PRAM*

Preliminary: Colored Compaction The colored compaction problem [4] is the following:

> Given n objects of m different colors, initially placed in a single source array, move the objects to m different destination arrays, one for each color. In this paper, we assume that the *space for the m destination arrays are preallocated*. We use the notation d_i to denote the number of objects colored i for $i \in [m]$.

Lemma 3. (Log*-time parallel algorithm for colored compaction [4]) *There is a constant $\epsilon > 0$ such that for all given $n, m, \tau, d_1, \ldots, d_m \in \mathbb{N}$, with $m \in O(n^{1-\delta})$ for arbitrary fixed $\delta > 0$, and $\tau \geq \log^* n$, there exists a parallel algorithm (in the arbitrary CRCW PRAM model) for the colored compaction problem (assuming preallocated destination arrays) with n objects, m colors, and $d_1, \ldots, d_m$ number of objects for each color, executing in $O(\tau)$ time on $\lceil n/\tau \rceil$ processors, consuming $O(n + \sum_{i=1}^{m} d_i) = O(n)$ space, and succeeding with probability at least $1 - 2^{-n^{\epsilon}}$.*

Preliminary: Parallel Static Hashing We will also rely on a parallel, static hashing algorithm [5,27], by Bast and Hagerup. The static parallel hashing problem takes n elements (possibly with duplicates), and in parallel creates a hash table of size $O(n)$ of these elements, such that later each element can be visited in $O(1)$ time. In our setting, we rely on the parallel hashing to suppress duplicate memory requests. Bast and Hagerup show the following lemma:

Lemma 4. (Log*-time parallel static hashing [5,27]) *There is a constant $\epsilon > 0$ such that for all $\tau \geq \log^* n$, there is a parallel, static hashing algorithm (in the arbitrary CRCW PRAM model), such that hashing n elements (which need not be distinct) can be done in $O(\tau)$ parallel time, with $O(n/\tau)$ processors and $O(n)$ space, succeeding with $1 - 2^{-(\log n)^{\tau/\log^* n}} - 2^{-n^{\epsilon}}$ probability.*

Construction We now present a construction that allows us to compile a P-parallel PRAM, where $P = M^{1+\delta}$ for any constant $\delta > 0$, into a $O(P/\log^* P)$-parallel oblivious NPRAM. The resulting NPRAM has $O(\log^* P)$ blowup in parallel runtime, and is optimal in total amount of work.

In the original P-parallel PRAM, each of the P processors does constant amount of work in each step. In the oblivious simulation, this can trivially be simulated in $O(\log^* P)$ time with $O(P/\log^* P)$ processors. Therefore, clearly the key is how to obliviously fetch a batch of P memory accesses in parallel with $O(P/\log^* P)$ processors, and $O(\log^* P)$

parAccess $\left(\{\mathsf{op}_p, \mathsf{vaddr}_p, \mathsf{wdata}_p\}_{p\in P}\right)$:

/ All steps can be executed in $O(\log^* P)$ time with $P' = O(P/\log^* P)$ processors with all but negligible probability. */*

1: Using the scratch bank as memory, run the parallel hashing algorithm in time $O(\tau) \in O(\log^* P)$, where $\tau := 2\log^* P$ on the batch of $P = M^{1+\delta}$ memory requests to suppress duplicate addresses. Denote the resulting set as S, and pad S with dummy requests to the maximum length P. This part fails with at most negligible probability (probability $2^{-(\log P)^2} + 2^{-P^\epsilon}$).

2: In parallel, assign colors to each memory request in the array S. For each real memory access $\{\mathsf{op}, \mathsf{vaddr}, \mathsf{wdata}\}$, its color is defined as $(\mathsf{PRF}_K(\mathsf{vaddr}) \mod M) + 1$. Each dummy memory access is assigned a random color. It is not hard to see that each color has no more than $\mathsf{max} := 2(P/M)$ requests except with negligible probability (probability $M \cdot \exp(-\frac{P}{3M}) = M \cdot \exp(-M^\delta/3)$).

3: Using the scratch bank as memory, run the parallel colored compaction algorithm to assign the array S to M preallocated queues each of size max (residing in the scratch bank). The algorithm fails with negligible probability (probability 2^{-P^ϵ}).

4: Now, each queue $i \in [M]$ contains max number of requests intended for bank i – some real, some dummy. Serve all memory requests in the M queues in parallel. Each processor $i \in [P']$ is assigned the k-th memory request iff $(k \mod P') = i$. Dummy requests incur accesses to the corresponding banks as well.
For each request coming from processor p, the result of the fetch is stored in an array $\mathsf{result}[p]$ in the scratch bank.

Fig. 6. Obliviously serving a batch of P memory requests using $P' := O(P/\log^* P)$ processors in $O(\log^* P)$ time. In Steps 1, 2, and 3, each processor will make exactly one access to the scratch bank in each parallel execution step—*even if the processor is idle in this step, it makes a dummy access to the scratch bank*. Steps 1 through 3 are always padded to the worst-case parallel runtime.

time. We describe such an algorithm in Fig. 6. Using a scratch bank as working memory, we first call the parallel hashing algorithm to suppress duplicate memory requests. Next, we call the parallel colored compaction algorithm to assign memory request to their respective queues—depending on the destination memory bank. Finally, we make these memory accesses, including dummy ones, in parallel.

Theorem 6. *Let* PRF *be a secure pseudorandom function, let $M = N^\epsilon$ for any constant $\epsilon > 0$ and recall that $N := N(\lambda)$ and $N \geq \lambda$. Let* PRAM *be a P-parallel RAM for $P = M^{1+\delta}$, for constant $\delta > 0$. Then, there exists an oblivious NPRAM simulation of* PRAM *with the following properties:*

- *The oblivious NPRAM consumes M banks each of which $O(N/M + P)$ words in size.*
- *If the underlying* PRAM *executes in t parallel steps, then the oblivious NPRAM executes in $O(t \log^* P)$ parallel steps utilizing $O(P/\log^* P)$ processors. We also say that the NPRAM has $O(\log^* P)$ blowup in parallel runtime.*
- *The total work of the oblivious NPRAM is asymptotically the same as the underlying* PRAM.

Proof. We note that our underlying PRAM can be in the EREW, CREW, common CRCW or arbitrary CRCW models. Our compiled oblivious NPRAM is in the arbitrary CRCW model.

We now prove security and costs separately.

Security Proof Observe that Steps 1, 2, and 3 in Fig. 6 make accesses only to the scratch bank. We make sure that each processor will make exactly one access to the scratch bank in every parallel step—even if the processor is idle in this step, it makes a dummy access. Further, Steps 1 through 3 are also padded to the worst-case running time. Therefore, the observable memory traces of Steps 1 through 3 are perfectly simulatable without knowing secret inputs.

For Step 4 of the algorithm, since each of the M queues are of fixed length max, and each element is assigned to each processor in a round-robin manner, the bank number each processor will access is clearly independent of any secret inputs, and can be perfectly simulated (recall that dummy request incur accesses to the corresponding banks as well). *Costs* First, due to Lemma 2, each of the M queues will get at most $2(P/M)$ memory requests with probability $1 - \text{negl}(N)$. This part of the argument is the same as Sect. 4. Now, observe that the parallel runtime for Steps 2 and 4 are clearly $O(\log^* P)$ with $O(P/\log^* P)$ processors. Based on Lemmas 4 and 3, Steps 1 and 3 can be executed with a worst-case time of $O(\log^* P)$ on $O(P/\log^* P)$ processors as well. We note that the conditions $M = N^\epsilon$ and $P = M^{1+\delta}$ ensure $\text{negl}(N) = \text{negl}(\lambda)$ failure probability. Specifically, the failure probability will be $O(2^{-(\log P)^2} + 2^{-P^\epsilon} + M \cdot \exp(-\frac{P}{3M}))$. □

6. Analysis of Deamortized Cuckoo Hash Table

In this section, we first describe the Cuckoo hash table of Arbitman et al. [2] and our modification of its parameters. Throughout this section, we follow [2] nearly verbatim. We describe the data structure of Arbitman et al. [2] in terms of a parameter $g(N)$. In the construction/proof of Arbitman et al. [2], the parameter $g(N)$ was set to be some function in $O(\log N)$. In contrast, in our construction/proof, we choose $g(N)$ to be any function $g(N) \in \omega(\log N)$.

The data structure uses two tables T_0 and T_1, and two auxiliary data structures: a queue, and a cycle-detection mechanism. Each table consists of $r = (1 + \epsilon)n$ entries for some small constant $\epsilon > 0$. Elements are inserted into the tables using two hash functions $h_0, h_1 : \mathcal{U} \to 0, \ldots, r - 1$, which are independently chosen at the initialization phase. We assume that the auxiliary data structures satisfy the following properties:

1. The queue is constructed to store $g(N)$, number of elements at any point in time. The queue of Arbitman et al. [2] was required to support the operations Lookup, Delete, PushBack, PushFront, and PopFront in worst-case $O(1)$ time (with $1 - 1/\text{poly}(N)$ probability over the randomness of its initialization phase). In our case, we require that the queue support the operations Lookup, Delete, PushBack, PushFront, and PopFront in worst-case $\widehat{O}(1)$ time (with all but negligible probability over the randomness of its initialization phase).
2. The cycle-detection mechanism is constructed to store $g(N)$, elements at any point in time. The cycle-detection mechanism of Arbitman et al. [2] was required to support the operations Lookup, Insert and Reset in worst-case $O(1)$ time (with $1 - 1/N$ probability over the randomness of its initialization phase). In our case, we require that the cycle-detection mechanism support the operations Lookup,

Insert and Reset in worst-case $\widehat{O}(1)$ time (with all but negligible probability over the randomness of its initialization phase).

An element $x \in \mathcal{U}$ can be stored in exactly one out of three possible places: entry $h_0(x)$ of table T_0, entry $h_1(x)$ of table T_1, or the queue. The lookup procedure is straightforward: when given an element $x \in \mathcal{U}$, query the two tables and if needed, perform lookups in the queue. The deletion procedure is also straightforward by first searching for the element, and then deleting it. Our insertion procedure is parametrized by a value $L = L(N)$, for any $L(N) \in \omega(1)$, and is defined as follows. Given a new element $x \in \mathcal{U}$, we place the pair $(x, 0)$ at the back of the queue (the additional bit 0 indicates that the element should be inserted to table T_0). Then, we take the pair at the head of the queue, denoted (y, b), and place y in entry $T_b[h_b(y)]$. If this entry is not occupied, we again take the pair that is currently stored at the head of the queue, and repeat the same process. If the entry $T_b[h_b(y)]$ is occupied, however, we place its previous occupant z in entry $T_{1-b}[h_{1-b}(z)]$ and so on, as in the above description of cuckoo hashing. After L elements have been moved, we place the current nestless element at the head of the queue, together with a bit indicating the next table to which it should be inserted, and terminate the insertion procedure.

We next restate Theorem 1:

Theorem 7. (Deamortized Cuckoo hash table: negligible failure probability version) *For any $g(N) \in \omega(\log N)$, there is an implementation of the above deamortized Cuckoo hash table of capacity s such that with probability $1 - \text{negl}(N) = 1 - \text{negl}(\lambda)$, each insertion, deletion, and lookup operation is performed in worst-case $\widehat{O}(1)$ time (not counting the cost of operating on the queue)—as long as at any point in time at most s elements are stored in the data structure. The above deamortized Cuckoo hash table consumes $O(s) + O(N^{\delta})$ space where $0 < \delta < 1$ is a constant.*

In the following, we describe the instantiation of the auxiliary data structures of Arbitman et al. [2] in terms of a parameter $g(N)$. In the construction/proof of Arbitman et al. [2], the parameter $g(N)$ was set to be some function in $O(\log N)$. In contrast, in our construction/proof we choose $g(N)$ to be any function $g(N) \in \omega(\log N)$.

The Queue We will argue that with overwhelming probability the queue contains at most $g(N)$ elements at any point in time. Therefore, we design the queue to store at most $g(N)$ elements, and allow the whole data structure to fail if the queue overflows. Although a classical queue can support the operations PushBack, PushHead, and PopFront in constant time, we also need to support the operations Lookup and Delete in $k(N)$ time, for any $k(N) \in \omega(1)$ (In Arbitman et al. [2], they required $k(N) \in O(1)$). One possible instantiation is to use $k := k(N)$ arrays $A_1, \ldots, A_k$ each of size N^{δ}, for some $\delta < 1$. Each entry of these arrays consists of a data element, a pointer to the previous element in the queue, and a pointer to the next element in the queue. In addition, we maintain two global pointers: the first points to the head of the queue, and the second points to the end of the queue. The elements are stored using a function h chosen from a collection of pairwise independent hash functions. Specifically, each element x is stored in the first available entry among $\{A_1[h(1, x)], \ldots, A_k[h(k, x)]\}$. For any element x, the probability that all of its k possible entries are occupied when the queue contains at

most $g(N)$ elements is upper bounded by $(g(N)/N^{\delta})^k$, which can be made negligible by choosing an appropriate k (in Arbitman et al. [2], this quantity could be made as small as $1/\operatorname{poly}(N)$ through appropriate choice of k).

The Cycle-Detection Mechanism As in the case of the queue, we will argue that with all but negligible probability the cycle-detection mechanism contains at most $g(N)$, elements at any point in time (in Arbitman et al. [2] this probability was $1-1/\operatorname{poly}(N)$). Therefore, we design the cycle-detection mechanism to store at most $g(N)$ elements, and allow the whole data structure to fail if the cycle-detection mechanism overflows. One possible instantiation is to use the above-mentioned instantiation of the queue together with any standard augmentation that enables constant time resets.

Note that in our case of negligible failure probability, the size of the queue and the cycle-detection mechanism are both bounded by $g(N) = \widehat{O}(\log N)$, instead of being bounded by $\log N$ as in [2]. It is not hard to see that as long as the auxiliary data structures do not fail or overflow, all operations are performed in time $\widehat{O}(1)$. Thus, our goal is to prove that with $1-\operatorname{negl}(N) = 1-\operatorname{negl}(\lambda)$ probability, the data structures do not overflow.

We continue with the following definition, which will be useful for the efficiency analysis.

Definition 7. Given a set $S \subseteq \mathcal{U}$ and two hash functions $h_0, h_1 : \mathcal{U} \to \{0, \ldots, r-1\}$, the cuckoo graph is the bipartite graph $G = (L, R, E)$, where $L = R = \{0, \ldots, r-1\}$ and $E = \{(h_0(x), h_1(x)) : x \in S\}$.

For an element $x \in \mathcal{U}$ we denote by $C_{S,h_0,h_1}(x)$ the connected component that contains the edge $(h_0(x), h_1(x))$ in the cuckoo graph of the set $S \subseteq \mathcal{U}$ with functions h_0 and h_1.

Similarly to [2], in order to prove Theorem 7, we require the following lemma:

Lemma 5. *For $T = f(N)$, where $f(N) = \omega(\log N)$, and $f(N) = o(\log N \log\log N)$, and $c_2 = \omega(1)$, we have that for any set $S \subseteq \mathcal{U}$ of size N and for any $x_1, \ldots, x_T \in S$ it holds that*

$$\Pr\left[\sum_{i=1}^{T} \left|C_{S,h_0,h_1}(x_i)\right| \geq c_2 T\right] \leq \operatorname{negl}(N) = \operatorname{negl}(\lambda),$$

where the probability is taken over the random choice of the functions $h_0, h_1 : \mathcal{U} \to \{0, \ldots, r-1\}$, for $r = (1+\epsilon)n$.

The proof of Lemma 5 will be discussed in Sect. 6.1.

Denote by $\mathcal{E}_1$ the event in which for every $1 \leq j \leq N/f(N)$, where $f(N) = \omega(\log N)$, and $f(N) = o(\log N \log\log N)$, it holds that

$$\sum_{i=1}^{f(N)} \left|C_{S,h_0,h_1}(x_{(j-1)\log(N)+i})\right| \leq c_2 f(N).$$

By using Lemma 5 and applying a union bound, we have that $\mathcal{E}_1$ occurs with probability $1 - \text{negl}(N)$.

We denote by $\mathsf{stash}(S_j, h_0, h_1)$ the number of stashed elements in the cuckoo graph of S_j with hash functions h_0 and h_1. Denote by $\mathcal{E}_2$ the event in which for every $1 \leq j \leq N/f(N)$, it holds that $\mathsf{stash}(S_j, h_0, h_1) \leq k$. A lemma of Kirsch et al. [28] implies that for $k = \omega(1)$, the probability of the event $\mathcal{E}_2$ is at least $1 - \text{negl}(N) = 1 - \text{negl}(\lambda)$.

The following lemmas prove Theorem 7:

Lemma 6. *Let π be a sequence of $p(N)$ operations. Assuming that the events $\mathcal{E}_1$ and $\mathcal{E}_2$ occur, then during the execution of π the queue does not contain more than $2f(N)+k$ elements at any point in time.*

Lemma 7. *Let π be a sequence of $p(N)$ operations. Assuming that the events $\mathcal{E}_1$ and $\mathcal{E}_2$ occur, then during the execution of π the cycle-detection mechanism does not contain more than $(c_2 + 1)f(N)$ elements at any point in time.*

The proofs of Lemmas 6 and 7 follow exactly as in [2], except the $\log N$ parameter from [2] is replaced with $f(N)$ in our proof and L (the time required per cuckoo hash operation) is finally set to $L(N) := c_2(N)(k(N) + 1)$ (so choosing $L(N) = g(N)$ for any $g(N) = \omega(1)$, we can find appropriate settings of $c_2(N), k(N)$ such that both $c_2(N), k(N) \in \omega(1)$ and $L(N) = c_2(N)(k(N) + 1)$).

6.1. *Proving Lemma 5*

As in [2], Lemma 5 is proved via Lemmas 8 and 9 below. Given these, the proof of Lemma 5 follows identically to the proof in [2].

Let $\mathbb{G}(N, N, p)$ denote the distribution on bipartite graphs $G = ([N], [N], E)$ where each edge is independently chosen with probability p. Given a graph G and a vertex v we denote by $C_G(v)$ the connected component of v in G.

Lemma 8. *Let $Np = c$ for some constant $0 < c < 1$. For $T = f(N)$, where $f(N) = \omega(\log N)$, and $f(N) = o(\log N \log\log N)$, and $c_2 = \omega(1)$, we have that for any vertices $v_1, \ldots, v_T \in L \cup R$*

$$\Pr\left[\sum_{i=1}^{T} |C_G(v_i)| \geq c_2 T\right] \leq \text{negl}(N),$$

where the graph $G = (L, R, E)$ is sampled from $\mathbb{G}(N, N, p)$.

We first consider a slightly weaker claim that bounds the size of the union of several connected components:

Lemma 9. *Let $Np = c$ for some constant $0 < c < 1$. For $T = f(N)$, where $f(N) = \omega(\log N)$, and $f(N) = o(\log N \log\log N)$, and $c_2' = O(1)$, we have that for any vertices $v_1, \ldots, v_T \in L \cup R$*

$$\Pr\left[\left|\bigcup_{i=1}^{T} C_G(v_i)\right| \geq c_2' T\right] \leq \text{negl}(N),$$

where the graph $G = (L, R, E)$ *is sampled from* $\mathbb{G}(N, N, p)$.

The proof of our Lemma 9 follows from Lemma 6.2 of [2]. Specifically, we observe that their Lemma 6.2 works for any choice of T (even though in their statement of Lemma 6.2, they require $T \leq \log N$. In particular, their Lemma 6.2 works for $T = f(N)$.

Next, the proof of our Lemma 8 can be obtained via a slight modification of the proof of Lemma 6.1 of [2]. Specifically, in their proof, they choose a constant c_3 and show that

$$\Pr\left[\sum_{i=1}^{T} |C_G(v_i)| \geq c_2' c_3 T\right] \leq \Pr\left[\left|\bigcup_{i=1}^{T} C_G(v_i)\right| \geq c_2' T\right] + \frac{(c_2' e)^{c_3} \cdot T^{2c_3+1}}{c_3^{c_3} \cdot n^{c_3}}.$$

By instead setting $c_3 = \omega(1)$, and using Lemma 9 to upper-bound $\Pr\left[\left|\bigcup_{i=1}^{T} C_G(v_i)\right| \geq c_2' T\right]$, we have that

$$\Pr\left[\sum_{i=1}^{T} |C_G(v_i)| \geq c_2' c_3 T\right] \leq \text{negl}(N),$$

since we choose $f(N) = o(\log N \log\log N)$. Again, note that setting $c_2 := c_2(N) \in \omega(1)$, we can find appropriate $c_2' := c_2'(N), c_3 := c_3(N)$ such that $c_3 = \omega(1), c_2' = O(1)$ and $c_2 = c_2' c_3$.

To get from Lemmas 8 to 5, we can go through the exact same arguments as [2] to show that $\mathbb{G}(N, N, p)$ is a good approximation of the Cuckoo graph distribution for an appropriate choice of p. Note that in our case of negligible failure probability, the size of the queue is bounded by $2f(N) + k(N)$ elements and the cycle-detection mechanism is bounded by $(c_2 + 1)f(N)$. Thus, again, by setting $g(N) \in \omega(\log n)$, we can find appropriate $f(N) \in \omega(\log n)$ and $k(N) \in \omega(1)$, that satisfy the restrictions on $f(N), k(N)$ required in the exposition above.

7. Conclusion

We define a new model for oblivious execution of programs, where an adversary cannot observe the memory offset within each memory bank, but can observe the patterns of communication between the CPU(s) and the memory banks. Under this model, we demonstrate novel *sequential* and *parallel* algorithms that exploit the "free obliviousness" within each bank, and asymptotically lower the cost of oblivious data accesses in comparison with the traditional ORAM [20] and OPRAM [7]. In the process, we propose novel algorithmic techniques that "leverage parallelism for obliviousness". These techniques have not been used in the standard ORAM or OPRAM line of work, and demonstrate interesting connections to the fundamental parallel algorithms literature.

Acknowledgements

We thank Srini Devadas, Ling Ren, Christopher Fletcher, and Marten van Dijk for helpful discussions. The first author is supported in part by an NSF CAREER Award #CNS-1453045, by a research partnership award from Cisco and by financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology. The third author is supported in part by NSF Grants #CNS-1514261 and #CNS-1652259. The fourth author is supported in part by NSF Grants #CNS-1314857, #CNS-1514261, #CNS-1544613, #CNS-1561209, #CNS-1601879, #CNS-1617676, an Office of Naval Research Young Investigator Program Award, a Packard Fellowship, a DARPA Safeware Grant (subcontractor under IBM), a Sloan Fellowship, Google Faculty Research Awards, a Google Ph.D. Fellowship Award, a Baidu Research Award, and a VMware Research Award. The fifth author is supported in party by NSF Grant #CNS-1161857.

References

[1] N. Alon, O. Goldreich, Y. Mansour. Almost k-wise independence versus k-wise independence. *Inf. Process. Lett.***88**(3), 107–110 (2003)

[2] Y. Arbitman, M. Naor, G. Segev. De-amortized cuckoo hashing: provable worst-case performance and experimental results, in *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5–12, 2009, Proceedings, Part I* (2009), pp. 107–118

[3] S. Bajaj, R. Sion. Trusteddb: a trusted hardware-based database with privacy and data confidentiality. *IEEE Trans. Knowl. Data Eng.***26**(3), 752–765 (2014)

[4] H. Bast, T. Hagerup. Fast parallel space allocation, estimation, and integer sorting. *Inf. Comput.***123**(1), 72–110 (1995)

[5] H. Bast, T. Hagerup. Fast and reliable parallel hashing, in *SPAA* (1991), pp. 50–61

[6] D. Boneh, D. Mazieres, R.A. Popa. Remote oblivious storage: making oblivious RAM practical (2011). http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf

[7] E. Boyle, K.-M. Chung, R. Pass. Oblivious parallel ram. https://eprint.iacr.org/2014/594.pdf

[8] K.-M. Chung, Z. Liu, R. Pass. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. *CoRR*. arXiv:1307.3699 (2013)

[9] C.W. Fletcher, M. van Dijk, S. Devadas. A secure processor architecture for encrypted computation on untrusted programs, in *STC* (2012)

[10] C.W. Fletcher, L. Ren, A. Kwon, M. Van Dijk, E. Stefanov, D.N. Serpanos, S. Devadas. A low-latency, low-area hardware oblivious RAM controller, in *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2–6* (2015), pp. 215–222. https://doi.org/10.1109/FCCM.2015.58

[11] C.W. Fletcher, L. Ren, A. Kwon, M. van Dijk, E. Stefanov, S. Devadas. RAW path ORAM: a low-latency, low-area hardware ORAM controller with integrity verification, in *IACR Cryptology ePrint Archive*, vol. 431 (2014)

[12] C.W. Fletcher, L. Ren, X. Yu, M. van Dijk, O. Khan, S. Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs, in *HPCA* (2014), pp. 213–224

[13] C. Gentry, K.A. Goldman, S. Halevi, C.S. Jutla, M. Raykova, D. Wichs. Optimizing ORAM and using it efficiently for secure computation, in *Privacy Enhancing Technologies Symposium (PETS)* (2013)

[14] C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, D. Wichs. Garbled ram revisited, in *Advances in Cryptology—EUROCRYPT 2014*, vol. 8441 (2014), pp. 405–422

[15] C. Gentry, S. Halevi, M. Raykova, D. Wichs. Garbled ram revisited, part i. Cryptology ePrint Archive, Report 2014/082, 2014. http://eprint.iacr.org/

[16] C. Gentry, S. Halevi, M. Raykova, D. Wichs. Outsourcing private ram computation. *IACR Cryptology ePrint Archive*, vol. 148 (2014)
[17] F. Ghanim, U. Vishkin, R. Barua. Easy PRAM-based high-performance parallel programming with ICE. *IEEE Trans. Parallel Distrib. Syst.* **29**(2), 377–390 (2018). https://doi.org/10.1109/TPDS.2017.2754376
[18] J. Gil, Y. Matias, U. Vishkin. Towards a theory of nearly constant time parallel algorithms, in *32nd Annual Symposium on Foundations of Computer Science (FOCS)* (1991), pp. 698–710
[19] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs, in *ACM Symposium on Theory of Computing (STOC)* (1987)
[20] O. Goldreich, R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM***43**(3), 431–473 (1996)
[21] M.T. Goodrich, D.S. Hirschberg, M. Mitzenmacher, J. Thaler. Fully de-amortized cuckoo hashing for cache-oblivious dictionaries and multimaps. *CoRR*. arXiv:1107.4378 (2011)
[22] M.T. Goodrich, D.S. Hirschberg, M. Mitzenmacher, J. Thaler. Cache-oblivious dictionaries and multimaps with negligible failure probability, in G. Even, D. Rawitz, editors, *Design and Analysis of Algorithms—First Mediterranean Conference on Algorithms, MedAlg 2012, Kibbutz Ein Gedi, Israel, December 3–5, 2012. Proceedings*. LNCS, vol. 7659 (Springer, 2012), pp. 203–218
[23] M.T. Goodrich, M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation, in *ICALP* (2011)
[24] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, R. Tamassia. Practical oblivious storage, in *ACM Conference on Data and Application Security and Privacy (CODASPY)* (2012)
[25] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation, in *SODA* (2012)
[26] S.D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, Y. Vahlis. Secure two-party computation in sublinear (amortized) time, in *ACM CCS* (2012)
[27] T. Hagerup. The log-star revolution, in *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13–15, 1992, Proceedings* (1992), pp. 259–278
[28] A. Kirsch, M. Mitzenmacher, U. Wieder. More robust hashing: cuckoo hashing with a stash, in *Algorithms—ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15–17, 2008. Proceedings* (2008), pp. 611–622.
[29] E. Kushilevitz, S. Lu, R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme, in *SODA* (2012)
[30] C. Liu, Y. Huang, E. Shi, J. Katz, M. Hicks. Automating efficient ram-model secure computation, in *IEEE S & P* (IEEE Computer Society, 2014)
[31] S. Lu, R. Ostrovsky. Distributed oblivious RAM for secure two-party computation, in *Theory of Cryptography Conference (TCC)* (2013)
[32] S. Lu, R. Ostrovsky. How to garble ram programs, in *EUROCRYPT* (2013), pp. 719–734
[33] S. Lu, R. Ostrovsky. Garbled ram revisited, part ii. Cryptology ePrint Archive, Report 2014/083, 2014. http://eprint.iacr.org/
[34] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, D. Song. Phantom: practical oblivious computation in a secure processor, in *CCS* (2013)
[35] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, D. Song. A high-performance oblivious RAM controller on the convey hc-2ex heterogeneous computing platform, in *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)* (2013)
[36] R. Meka, O. Reingold, G.N. Rothblum, R.D. Rothblum. Fast pseudorandomness for independence and load balancing—(extended abstract), in *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8–11, 2014, Proceedings, Part I* (2014), pp. 859–870
[37] R. Ostrovsky, V. Shoup. Private information storage (extended abstract), in *ACM Symposium on Theory of Computing (STOC)* (1997)
[38] R. Pagh, F.F. Rodler. Cuckoo hashing. *J. Algorithms***51**(2), 122–144 (2004)
[39] L. Ren, X. Yu, C.W. Fletcher, M. van Dijk, S. Devadas. Design space exploration and optimization of path oblivious RAM in secure processors, in *ISCA* (2013), pp. 571–582
[40] J.P. Schmidt, A. Siegel, A. Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math.***8**(2), 223–250 (1995)

[41] E. Shi, T.-H. Hubert Chan, E. Stefanov, M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost, in *ASIACRYPT* (2011)
[42] E. Stefanov, E. Shi. Oblivistore: high performance oblivious cloud storage, in *IEEE Symposium on Security and Privacy (S & P)* (2013)
[43] E. Stefanov, E. Shi, D. Song. Towards practical oblivious RAM, in *NDSS* (2012)
[44] E. Stefanov, M. van Dijk, E. Shi, T.-H.H. Chan, C. Fletcher, L. Ren, X. Yu, S. Devadas. Path ORAM: an extremely simple oblivious ram protocol, in *ACM CCS* (2013)
[45] U. Vishkin. Can parallel algorithms enhance seriel implementation? *Commun. ACM* **39**(9), 88–91 (1996)
[46] U. Vishkin. Using simple abstraction to reinvent computing for parallelism. *Commun. ACM* **54**(1), 75–85 (2011)
[47] X.S. Wang, T.-H.H. Chan, E. Shi. Circuit ORAM: on tightness of the Goldreich–Ostrovksy lower bound. http://eprint.iacr.org/2014/672.pdf
[48] X.S. Wang, Y. Huang, T.-H.H. Chan, A. Shelat, E. Shi. Scoram: oblivious ram for secure computation. http://eprint.iacr.org/2014/671.pdf
[49] P. Williams, R. Sion. Usable PIR, in *Network and Distributed System Security Symposium (NDSS)* (2008)
[50] P. Williams, R. Sion. SR-ORAM: single round-trip oblivious ram, in *ACM Conference on Computer and Communications Security (CCS)* (2012)
[51] P. Williams, R. Sion, B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage, in *CCS* (2008)
[52] P. Williams, R. Sion, A. Tomescu. PrivateFS: A parallel oblivious file system, in *CCS* (2012)
[53] X. Yu, S.K. Haider, L. Ren, C.W. Fletcher, A. Kwon, M. van Dijk, S. Devadas. Program: dynamic prefetcher for oblivious RAM, in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13–17, 2015* (2015), pp. 616–628